INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Integrating formal verification methods of quantitative real-time properties into a development environment for robot controllers

Muriel Jourdan

N° 2540

Avril 1995

PROGRAMME 4

ISSN 0249-6399

Integrating formal verification methods of quantitative real-time properties into a development environment for robot controllers

Muriel Jourdan

Programme 4 — Robotique, image et vision
Projet Bip

Rapport de recherche n°2540 — Avril 1995 — 29 pages

Abstract: In this paper we describe our experience with a development environment for robot controllers, which provides the user with formal verification functionalities. We study how to augment these functionalities by also allowing formal verification of quantitative real-time properties. Our approach is based on the timed extension of a synchronous language, named Timed-Argos, and on a symbolic model-checking tool named Kronos for the real-time temporal logic TCTL. We illustrate this approach by a real example taken from the area of autonomous vehicles, which poses some challenges on the applicability of the theory and finally, we discuss some possible solutions. This *large-scale* real application is also an opportunity to identify new research directions in the area of formal verification.

Key-words: formal verification, real-time systems, synchronous langages, temporal logics, robotic

(Résumé : tsvp)

Unité de recherche INRIA Rhône-Alpes
46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1 (France)
Téléphone : (33) 76 57 47 77 – Télécopie : (33) 76 57 47 54

Vérifier des propriétés quantitatives temporelles dans un environnement de développement d'applications robotique

Résumé :

Nous décrivons dans ce rapport comment intégrer des méthodes de vérification formelle de propriétés quantitatives temporelles à un environnement d'aide au développement d'applications robotique nommé ORCCAD. Notre approche est basée sur l'utilisation d'une extension temporelle du langage synchrone Argos et d'un outil de vérification symbolique nommé Kronos. Nous illustrons cette approche sur un exemple réel de conduite autonome de véhicules. Cette expérimentation est pour nous l'occasion de valider une approche formelle et de découvrir de nouveaux besoins afin d'orienter les recherches menées dans le domaine de la vérification formelle des systèmes temps-réels.

Mots-clé : vérification formelle, systèmes temps-réels, langages synchrones, logiques temporelles, robotique

Introduction

Motivations

The aim of this paper is to show how formal methods of verification could be used in a real application area such as in robotics, in order to check time dependent properties. More precisely, we bring to the fore front some particular needs inside a development environment for robot controllers named ORCCAD [SECK93], and we show how to fulfil them by using a language named Timed Argos [JMO93] and a symbolic verification tool named Kronos [HNSY92].

This experiment in a real application area is an opportunity of making a qualitative analysis — adequation of the approach to the needs, integration into the existing environment, ... — and a quantitative one — Argos compiler and Kronos tool performances — of both Argos and Kronos. This is also a good way to identify possible new research directions in the area of formal verification.

Context

Robotic systems are hybrid systems operating in real-time and handling events as well as "continuous computations". Reliable and easy programming of these systems requires a systematic method for the specification of robotic applications, formal verification of their execution from a continuous and discrete-time point of view and efficient implementation over the target architecture.

The ORCCAD system (Open Robot Controller Computer-Aided Design) proposes a coherent approach from a high-level specification down to its implementation by harmoniously integrating discrete and continuous aspects. It is based on automatic control theory for the design and analysis of the control law, and on reactive systems theory for the discrete events aspect.

In this paper, we are interested only on the discrete events aspects. More precisely we address the problem of formal verification of discrete events robot controllers inside ORCCAD. These controllers are currently represented as boolean automata, which are obtained by translating ORCCAD specification formalisms into a subset of the synchronous language Esterel [BG92], whose semantics are expressed in terms of boolean automata. Thus, it is possible to use inside ORCCAD the verification tool Auto [dSV89] which has a friendly interface with Esterel. This tool could be used to build abstract views of the global controller which could be either directly observed (if the resulting automaton is small) or compared to another automaton which expresses the property. This approach is well-adapted to check global properties like the absence of deadlock.

However, this method does not allow to verify quantitative real-time properties although a lot of explicit delays appear in the controller specification (durations, timeouts). For instance, it would be interesting to be able to prove that the execution of a robotic application is time-bounded or that the time-lag between the starting points of two execution laws is bounded.

A way of proving such properties, consists in modeling robot controllers as Timed-Graphs [AD90] — boolean automata extended with time counters — and using a symbolic verification tool. The synchronous language Argos [Mar92] has been first inspired from Statecharts [Har87]. Its semantics, like Esterel, was expressed in terms of boolean automata. It has been recently extended with a time construct and its semantics could now be expressed in terms of Timed-Graphs. The Kronos tool which implements a symbolic model-checking algorithm for the real-time temporal logic TCTL [ACD90] is interfaced with Timed-Argos.

Outline of the paper

The remainder of the paper is organized as follows. In the first section we present some needs of time dependent formal verifications inside ORCCAD through an overview of this system. The second section is devoted to the presentation of the Timed Argos language and the Kronos symbolic verification tool. In the third section, we describe how to integrate these formalisms and tools into ORCCAD in order to be able to check time dependent properties. An example taken from the area of autonomous vehicles is presented to illustrate the integration principles. Finally, in the last section we discuss qualitative and quantitative analysis of the experiments previously described.

1 An overview of ORCCAD

The basic entity in ORCCAD [SECK93] is the concept of *Robot-Task* (RT in the sequel). Its function is to specify and implement simple robotic actions. Complex actions are obtained by composing RTs using different kinds of "synchronization" operators, the final result being called a *Robot-Procedure* (RP in the sequel). For instance, suppose that the mission consists in automatically parking a car. First, the car must search for a free parking space and then it should park itself. If these two steps are not too complex, the mission could be specified by using two RTs which are run in sequence in the nominal execution of the RP. We first present the RT concept, then we present how to combinate these basic objects inside a RP in order to specify and implement a robotic application.

1.1 The Robot-Task concept

RT consists of algorithmical aspects relating to the control law and logical aspects relating to the local behavior associated with a set of events which may occur before or during the control law execution. This local behavior is predefined. The execution of the control law begins as soon as a set of *preconditions* is satisfied. The waiting time for each precondition to be satisfied can be bounded. If it elapses, the RT is stopped. The execution of the control law can be interrupted if "problems" are detected. It ends normally either after the satisfaction of a set of *postconditions* or because of the specified RT duration is exceeded.

In order to specify this local behavior, the user gives :

- a set of precondition events. A waiting time limit could be associated with each event.
- a set of "exception" events. Three kinds of exceptions could be specified:
  - global exception : it interrupts all the RTs used in the application.
  - local exception : it interrupts only the RT. We will see later how local exceptions are handled by RPs.
  - local change : it does not interrupt a RT but indicates a modification of parameters which appear in the control law.
- a set of postcondition events. A waiting time limit could be associated with each event.
- a duration.

This specification is automatically translated into an Esterel program in which all quantitative delays have been transformed into logical timeout events. This program is compiled into a boolean automaton. It has been proved in [Kap94], that this automatic translation guarantees two important properties:

- it satisfies a liveness property, i.e. a successful termination of the RT can be reached from any state of its evolution;
- it satisfies a safety property, i.e. any global exception is appropriately handled by emission of a specific event leaving the system in a safe situation.

1.2 The Robot-Procedure formalism to combine Robot-Tasks

With the RP formalism one can specify in a structured way a logical and temporal arrangement of RTs in order to achieve an objective in a context dependent and reliable way, providing predefined correction actions in the case of unsuccessful executions of RTs.

The user specifies a similar behavior to the RT one.

- a set of precondition events. A waiting time limit could be associated with each event.
- a set of "exception" events. Two kinds of exceptions could be specified:
  - global exception : it interrupts all the RPs used in the application.
  - local exception : it interrupts only the RP.
- a set of postcondition events. A waiting time limit could be associated with each event.

The user also specifies a *main* program and a set of *exception* programs. The main program corresponds to the nominal execution of the robot and is started as soon as the preconditions are satisfied. The exception programs are associated with local exceptions handled in the RTs or RPs (the RP formalism is recursively defined) used by the main program. These two kinds of programs are specified using composition operators of RTs or RPs already defined. For instance, it is possible to express the sequence of two RTs (or RPs) or their "parallel" execution: they are started at the same instant, and the end of the construction occurred when the two operands end. It is also possible to take into account external conditions with a kind of `if_then_else` structure. The waiting time for the condition to be satisfied could be bounded.

The whole controller specified by a RP can be translated into an Esterel program (delays are transformed into logical timeout events) in order to be represented by a boolean automaton. As a consequence, it is possible to check some properties like deadlock absence or conformity of the RP behavior with respect to mission constraints. Contrary to the RT behavior, which is proved correct by construction, the user is in charge of the RP behavior verification, which should be done with the Auto tool. The properties he could express are not time-dependent. For instance, he could not check that the execution time of a RP is always lower than a given value.

The aim of the work presented in this paper is precisely to withdraw this impossibility. The idea is to replace the translation into Esterel with a translation into Timed Argos in order to keep the values of the delays in the program and to use the symbolic verification tool Kronos on the resulting timed graph.

2 The Timed-Argos language and the Kronos tool

The Argos synchronous language [Mar92] was first inspired by Statecharts [Har87]. It provides the user with a set of operators that can be applied to elementary automata components to build more complex systems. These operators include parallel composition and hierarchic composition. The Argos semantics is expressed in terms of boolean automata. Argos has been recently extended with a delay construction [JMO93] which allows to express watchdogs and timeouts easily. The resulting language is named Timed-Argos. Its semantics is expressed in terms of Timed-Graphs, which are automata extended with real clocks. With this kind of program models, one can check real-time quantitative properties, using symbolic verification tools such as Kronos. Kronos is a tool which implements a symbolic model-checking algorithm [HNSY92] for the real-time temporal logic TCTL of Timed-Graphs [ACD90].

In this section, we first present the Argos language. Next we present its temporal extension. Finally, we present the Kronos verification tool.

2.1 The synchronous language Argos

A more detailed presentation of Argos could be found in [JM94].

In Argos, a simple system is described directly as an automaton by giving the set of states and transitions explicitly. When it is more complex, it can be described as a combination of several such components, using operators like the parallel composition, which express how they should be "connected" in order to communicate and participate to the global behavior of the system.

In the parallel composition, for instance, two or more components evolve in parallel. They can communicate with each other by exchanging signals: if an input signal of a components and an output signal of another one have the same name, the two components are forced to communicate in this way. The communication is the *synchronous broadcast*. It is non blocking (unlike the *rendez-vous* mechanism, for instance). A component can always output signals. The other ones can always react to them because they are *reactive*, but some of them have a null reaction (they do not change states nor output signals).

Figure 1 is an ARGOS program using five automata to describe a modulo-8 `a`-counter with initialization and interruption facilities. It satisfies the following specification: once `start` is received, emit `mod8a` every 8 `a`'s. Stop counting the occurrences of `a` when either 6 time units have elapsed since the `start` signal, or `stop` has occurred.

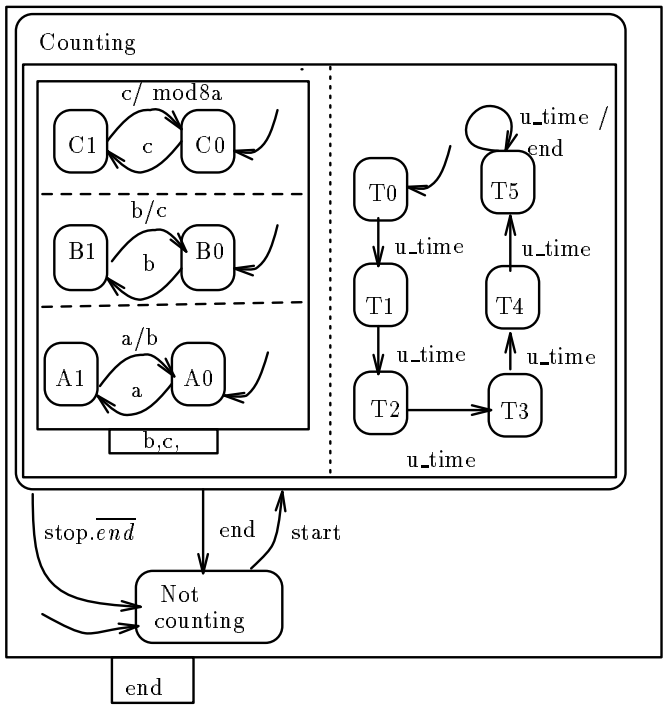

Figure 1: Argos program for the modulo-8 `a`-counter

Rounded-corner boxes are automaton states; arrows are transitions; rectangular boxes are used for unary operators (see below). A set of states and transitions which are connected together constitutes an automaton. The five basic components of the program have the following sets of states: {`Counting, Not counting`}, {`A0, A1`}, {`B0, B1`}, {`C0, C1`} {`T0, T1, T2, T3, T4, T5`}.

In the automata, the input part and the output part of a transition label are separated by a slash (example: c/end). Negation is denoted by overlining, and conjunction is denoted by a dot (example $stop.\overline{end}$). When the output set is empty, it can be omitted. The initial state is designated by an arrow without an associated source. For example, the arrow connected on the right to `C0`. States are named, but names should be considered only as labels to help the programmer.

An arrow can have several labels which stand for several transitions. The different labels of a same arrow are separated by a comma. Some labels have to be considered as abbreviations. For instance, the `end` label stands for both `end.stop`, `end.`$\overline{stop}$.

The automaton whose states are `Counting` and `Not counting` is said to be *refined* in its `Counting` state. The refining subprogram is built with the four other automata, which are put *in parallel* (In figure 1 they are shown separated by dashed lines). The outermost box, with an attached subbox which contains `end`, is the graphical syntax for a unary local signal declaration operator. The box defines the scope in which the signal `end` is known. This signal is used as input by the refined automaton; it is used as output by the timer component (vertical automaton in the refined state): a communication will take place between the two. Another such unary operator is used in the program, in order to limit the scope of the signals `b, c` to the program constituted by the three unrefined automata (drawn horizontally).

The interface of the global program is defined as follows: all signals which appear in a left-hand (resp. right-hand) side of a label, and are not declared to be local to some part of the program are *global inputs* (resp. *global outputs*).

We give here the intuitive semantics of the operators, by explaining the behavior of the counter.

First, observe the three (horizontal) automata embedded in a parallel structure, and the operator which defines the scope of `b` and `c`. This constitutes a subprogram whose only input is `a`, and whose only output is `mod8a`. The global behavior of this subprogram is defined by: the global initial state is `C0,B0,A0`; when it has reacted to input `a` n times, the program is in state $\mathtt{C}_k,\mathtt{B}_j,\mathtt{A}_i$, where $i + 2j + 4k = n \bmod 8$; `mod8a` is output every 8 `a`'s.

This behavior is achieved by connecting three one-bit counters. The first one (A) reacts to external input `a`, and triggers the second one (B) with signal `b`, every two `a`'s. The second one, reacting to `b`, triggers the third one (C) with `c`, every two `b`'s. The third one outputs `mod8a` every two `c`'s. The communication being *synchronous*, a reaction "in three steps", to which the three bits participate, is indeed *one* transition in the global behavior (reaction to `a` from `C0,B1,A1` to `C1,B0,A0`).

Second, observe the timer component: it evolves apart from the three previous automata, counting the occurrences of `u_time`. It outputs `end` 6 units of time after its initialization.

Finally, these four automata in parallel refine the `counting` state of a two-state automaton. The `counting state` is reached when `start` occurs. As a consequence, the refining subprogram is started in its initial state. It is killed when `stop` occurs:

it is an *interruption*. Leaving a refined state may also be done by *self-termination* of the refining subprogram. This is the case in our example when `end` is output.

It is outside the scope of this paper to give the formal semantics of Argos [Mar92]. However, in order to understand the timed extension of Argos we have to present how this semantics is defined. The Argos semantics is *syntax-directed*. It describes the behavior of a program by defining a function $\mathcal{S}$ such that $\mathcal{S}(automaton) = automaton$ and $\mathcal{S}(P1\ op\ P2) = \mathcal{F}_{op}(\mathcal{S}(P1), \mathcal{S}(P2))$.

An interesting characteristic of Argos semantics is that it is easy to rely model information on source program. Indeed, a model state (resp. a model transition) is a set of "active" states (resp. "fired" transitions) of the source program automata. We will see later how to use this information in the verification process.

2.2 Timed-Argos to describe Timed-Graphs

2.2.1 A form of program models :Timed-Graphs

Timed Graphs are automata extended with a finite set of real-valued clocks. They are of the form (Q, q_0, T, F, X) where Q is a set of *nodes*, q_0 is the *initial node*, X is a set of real variables called *clocks*, $T \subseteq Q \times C(X) \times L \times R(X) \times Q$ is a set of *transitions*. $C(X)$ is a set of boolean conditions built from the variables in X following the grammar $c ::= x\ op\ k$, where k is a nonnegative integer constant and op belongs to $\{\leq, \geq, <, >, =, \neq\}$. L is a set of labels. $R(X)$ is a set of *reset actions*. A *reset action* $r \in R(X)$ is a subset of X which contains the variables to be reset when the transition is taken. The other variables are unchanged. $F : Q \rightarrow C(X)$ gives the labelling of nodes by invariant properties.

The semantics of Timed Graphs is defined in terms of Mealy Machines. States are of the form $(q, \vec{v})$ where q is a node of the given Timed Graph and $\vec{v}$ is a valuation of its clocks. Transitions are labelled either by an element of L or by a real value which denotes the time passing. This semantics is based intuitively on the following principles: a) The initial state is equal to $(q_o, \vec{0})$ ($\vec{0}$ is the valuation which associates 0 with each clock); b) Transitions are instantaneous; c) Time elapses in nodes; d) The node invariants have to be always satisfied.

For instance, if we consider the Timed Graph given in figure 2,
$(A,0) \xrightarrow{3.2} (A,3.2) \xrightarrow{l1} (B,0) \xrightarrow{5} (B,5) \xrightarrow{l2} (A,5)$
is a possible execution sequence.

The need for a high level language to describe Timed Graphs is obvious since we could not manage to describe a complex system by a Timed Graph directly. Timed Argos offers high level constructions to describe an interesting subclass of Timed Graphs in a compositional way.

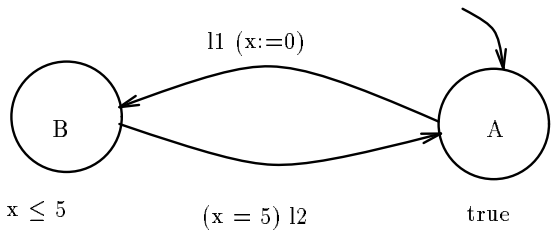

Figure 2: An example of Timed Graph

2.2.2 Timed Argos : a temporal extension to Argos

Timed Argos [JMO93] provides a delay construction to express watchdogs and timeouts. For instance, using this new construction the counter system could be described by the program given in figure 3.

The timer component used by the program given in figure 1 has been replaced by a one state automaton. This state is said to be temporized by the delay 6. It has an outgoing transition whose label is replaced by a square box. This is the *timeout* transition. The intuitive semantics is as follows : once a temporized state is entered, it *must* be left before the indicated amount of time has elapsed. The program can leave the temporized state by taking a "normal" transition (if such transitions exist), or it has to take the special timeout transition, when the delay expires. Automata with temporized states are called *timed automata*. They are the basic objects of Timed Argos.

Temporized states in a Timed Argos program are translated into clocks in the corresponding Timed Graph.

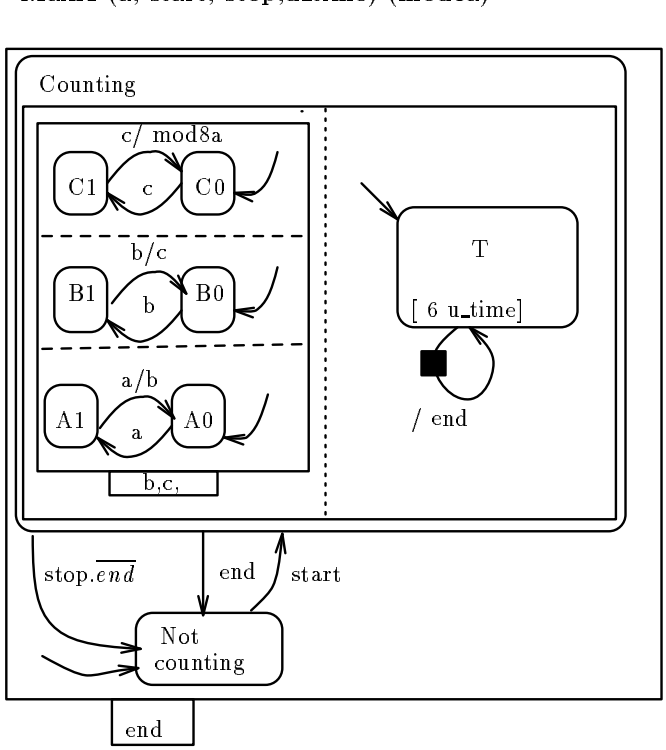

Figure 3: Timed Argos program for the modulo-8 a-counter

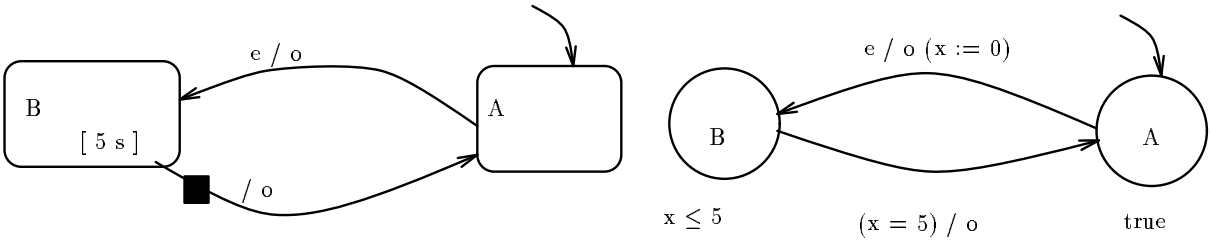

Figure 4: Translation of a timed automaton into a Timed Graph

The idea of the new semantics function $\mathcal{S}^t$ [JMO93] (which translates all Timed Argos programs into Timed Graphs) is the following: $\mathcal{S}^t(\textit{timed automaton}) =$

timed graph and $\mathcal{S}^t(P1\ op\ P2) = \mathcal{F}^t_{op}(\mathcal{S}^t(P1), \mathcal{S}^t(P2))$. The transition labels of the Timed Graphs built from Timed Argos and the transition labels of Argos automata have the same form. We give in figure 4 an example of the translation of a timed automaton into a Timed Graph.

2.3 The Kronos Verification tool

The Kronos tool implements a symbolic model-checking algorithm for TCTL [ACD90] (a real-time extension of the branching-time logic CTL) on Timed Graphs [HNSY92]. It means that the property is expressed by a TCTL formula and that Kronos computes the set of states of the Mealy Machines associated with the Timed Graph which satisfy it. The property is satisfied if and only if the initial state belongs to this set. The algorithm implemented by Kronos is symbolic, since the Mealy Machine associated with the Timed Graph is never computed, but rather represented implicitly.

A TCTL formula ϕ is built following the grammar :

$$\phi ::= p | \neg\phi | \phi \wedge \phi | \forall\exists_{\#c}\phi | \exists\forall_{\#c}\phi | \forall\forall_{\#c}\phi | \exists\exists_{\#c}\phi$$

$\#$ belongs to $\{<, \leq, >, \geq\}$. c is an integer value. p is a property of *states* (i.e. nodes and valuation of the clocks), and can be identified to the set of states where it is true. The set can be given in extension, but it is usually described by using a function which builds *state properties* out of *transition properties*. For instance, `enable`(l) computes the set of states q such that there exists at least one transition sourced in q and labelled by l. These functions are expressed in terms of the Timed-Graph states or transitions. Given the fact that the Argos semantics is such that it is easy to rely model information on source program, it is possible to use specific functions more intuitively for the programmer. For instance, it is possible to characterize a set of states in the model of a Timed Argos program with the following function: `InState`(q) where q is the name of an automaton state in the source program. The Argos compiler keeps enough information for the Kronos tool to be able to evaluate this function.

Let us illustrate the semantics of TCTL with the following example: $\exists\exists_{<4}\{q\}$. A node q' satisfies this formula if and only if there exists one execution sequence from q' such that a state satisfying q is reached before 4 units of time. It expresses the *possibility* to reach q' before 4 units of time. Some formulas do not have temporal restrictions (given by the $\#c$ expression): $\forall\exists\phi$ expresses that ϕ will be satisfied *eventually*, i.e. for each execution sequence from q' there exists a state satisfying ϕ.

$\forall\forall\phi$ expresses that ϕ is an *invariant* property and $\exists\forall\phi$ is satisfied if and only if there exists one execution of the program on which ϕ is always satisfied.

3 Using Timed-Argos and Kronos inside ORCCAD

3.1 Principles

In this section we give only an intuitive idea of how we propose to integrate verification methods of quantitative real-time properties into ORCCAD. These principles will be illustrated below while describing an experiment in the area of autonomous vehicles.

- First, we translate RTs and RPs into Timed Argos, in order to represent the controller behavior of the robotic application by a Timed-Graph (and not by a boolean automaton). A minimization tool is used on the Timed-Graph computed by the Argos compiler before using Kronos. This minimization is effective since, first, the behaviors we specify are such that a lot of transitions lead to the same state with different labels; and second, the properties we consider could be easily expressed in terms of the Timed-Argos source program states (see below). As a consequence, labels could be abstract since they are not related to the properties. This reduction phase is achieved by using the minimization tool named Aldebaran [Fer88].
- Second, we express the property in TCTL and use the Kronos tool to check it.

We are now going to illustrate this principle of integration with a real example taken from the area of autonomous vehicles.

3.2 An experiment in the area of autonomous vehicles

3.2.1 Informal specification of the example

Our long-term objective is to specify, validate and implement a virtual "train" of electric vehicles: each vehicle is expected to closely follow the previous one automatically using dedicated sensors — a vision approach is used to locate the previous vehicle in distance and angle — and a computerized control system. The first vehicle would be the only one with a human driver. This work is part of the *Praxitèle Project* (as in [PDDM94]), the ambitious program by the French government to develop a self-service public transport system using small electric vehicles. We are currently working with only two vehicles.

The application we described should respect the following specification. When it is started, the driven car should signal when it is ready. Then, the undriven car, which should be in the automatic mode, tries to catch the video signal to locate the first car. When it is done, the nominal execution expected is that the undriven car follows the driven car until the application is stopped. An exception program must be started if the video signal is lost during this nominal execution. It has to stop the undriven car as quick as possible. The driven car is supposed to come back and the "train" reformed. A lot of problems could hamper this execution: physical damages of crucial components, activation of the manual mode, mechanical stops, ...

The types of quantitative real time properties we would like to check on this example is :

- the application is time bounded.
- the time-lag between the starting points of the wheel execution law and the motor execution law is bounded.
- the time-lag between the detection of the signal loss detection and the complete stopping of the car is bounded.

3.2.2 Translation into the Robot-Procedure formalism

We translate this informal specification into a RP. Its nominal execution consists of an infinite loop whose body begins with the test of an external condition which indicates that the driven car is ready. Whenever it is satisfied a RP named RP_guarded_move is started. This second RP aims to control the second car when it follows the first one and to handle the loss of the video signal between the two vehicles.

Before beginning this nominal execution, a set of three preconditions must be satisfied. The initialization phase (motors, sensors, ...) must have been made without detecting errors. The active mode must be activated and the "human" supervisor has to give the start order. The nominal execution of this main RP is stopped in two cases, either the supervisor gives a stop order or the manual mode is activated. In the first car, the RP ends normally; in the second one it is interrupted by a global exception.

The exact specification of this main RP is the following one :

```
Name          : main
Preconditions : ok_init [waiting time limit : 30 ms]
              : auto_mode [waiting time limit : 30 ms]
              : start [waiting time limit : 5 mn]
Postconditions : stop [waiting time limit : 60 mn ]
```

```
Global exceptions  : auto2man
Local exceptions  : none
Nominal execution  :
               Loop
                    wait first_car_ready [waiting time limit : 5 mn]
                    start(RP_guarded_move)
               endLoop
Exception treatments  : none
```

RP_guarded_move is built from the parallel composition of three basic robotic actions specified as three independent RTs. The first two, RT_sens_loc and RT_sens_dir, respectively control the electric motor and the wheel of the car according to sensors information. The last one, named RT_brake, controls the foot-brake of the car. It is not always active, since in most of the cases the engine_braking is sufficient to stop the vehicle. The fisrt two RTs detect a local exception when the video signal between the two cars is lost. The RP RP_guarded_move handle this situation by starting an "emergency" procedure named RP_parking.

The exact specification of RP_guarded_move is the following one.

```
Name  : RP_guarded_move
Preconditions  : none
Postconditions  : none
Global exceptions  : none
Local exceptions  : none
Nominal execution  :
               Parallel
                    start(RT_sens_loc)
                    ||
                    start(RT_sens_dir)
                    ||
                    Loop
                         if  more_brake then start(RT_brake)
                    endLoop
               endParallel
Exception treatments  :
               if  ( signal_lost_RT_sens_loc or signal_lost_RT_sens_dir )
                    then start(RP_parking)
```

We could notice that the nominal execution of the RP RP_guarded_move never ends, since it is built from three parallel components in which one is an infinite loop. The only situation where RP_guarded_move ends is when a loss of signal video is detected and when RP_parking ends.

We are now going to describe one RT. We choose RT_sens_loc. We remind the reader that a RT consists of algorithmical aspects relating to the control law (in our example, it determines the speed of the vehicle according to some sensor information which gives for instance the distance between the two vehicles) and logical ones which describe the discrete control of the robotic actions. We are interested in our paper only with the second aspects.

The control law of RT_sens_loc could be started if and only if the initialization phase has been made without detecting errors, and if the execution law which controls the wheel of the car is already activated. The detection of a mechanical problem in the motor of the car leads to a emergency stop of all the application. Moreover, if the speed value computed by the control law is too high, some parameters of this control law must be changed to compute a new speed value.

The exact specification of this RT is the following one:

`Name` : RT_sens_loc
`Preconditions` : motor_ok [`waiting time limit` : 5 ms]
: wheel_started [`waiting time limit` : 5 ms]
`Postconditions` : `none`
`Duration` : `undefined`
`Global exceptions` : motor_pb
`Local exceptions` : signal_lost
`Local changes` : speed_overload

RT_sens_dir is very similar to RT_sens_loc, the only difference is that it has no equivalent to the precondition wheel_started (it has only one precondition which indicates that the initialization phase was correctly done).

RT_brake and RP_Parking are defined in the appendix of the paper.

3.2.3 Translation into Timed-Argos

First, the Kronos verification tool handles only one time unit: each clock increases at the same speed. We are thus obliged to translate each delay which appears in the specification into milli-seconds.

We begin to show how to translate a RT into Timed-Argos.

A Timed-Argos RT component has three states which correspond to the RT current status: while waiting for the precondition to be satisfied, active (the execution law is alive) and finished. The first state is refined by $i+1$ automata in parallel where i is the number of precondition specified in the RT. Indeed, one automaton is associated with each precondition and one controller is added which detects that each precondition is satisfied. If the RT has a duration, the "alive" state is temporized. In all cases, it is refined by $j + k + l + m + 1$ automata in parallel where:

- j is the number of local change exceptions;
- k is the number of local exceptions;
- l is the number of global exceptions;
- m is the number of postconditions;

One controller is added which detects that each postcondition is satisfied.

For instance, the translation of the RT RT_sens_loc into Timed Argos leads to the program given in the figure 5.

In order to show how a RP could be translated into Timed Argos, we used the Argos procedure call mechanism. With this mechanism it is possible to re-use an Argos component in a program with events renaming possibilities. Procedure calls of an Argos program are expanded before the compilation process.

The figure 6 is the Timed Argos program associated with RP_guarded_move. We could see that the nominal state is refined by a parallel composition of three components: the two RT which respectively control the motor and the wheel of the car; and a third component which handles the foot_brake when it is necessary.

3.2.4 Verification process

We are now going to show on this particular robotic application how to check time dependent properties.

Our first example consists in proving that the maximum execution time of the application is always lower than 70 minutes. This property seems to be true since the specification of the main RP indicates that the maximum waiting time for the preconditions to be satisfied (`max_prec` in the sequel) is equal to 5 minutes and that the minimum waiting time for the postconditions to be satisfied (`min_post` in the sequel) is equal to 60 minutes. Nevertheless, nothing indicates that the program which implements this specification has been correctly constructed. Moreover, the maximum execution time of a RP could be lower than a value T such as $T <$`max_prec` + `min_post`. This is not the case in our example, but it could be true for other ones

RT_sens_loc (motor_ok, wheel_ok,speed_overload, motor_pb, signal_lost)

(GExc_RT_sens_loc, LExc_RT_sens_loc, LC_speed_overload, signal_lost_RT_sens_loc)

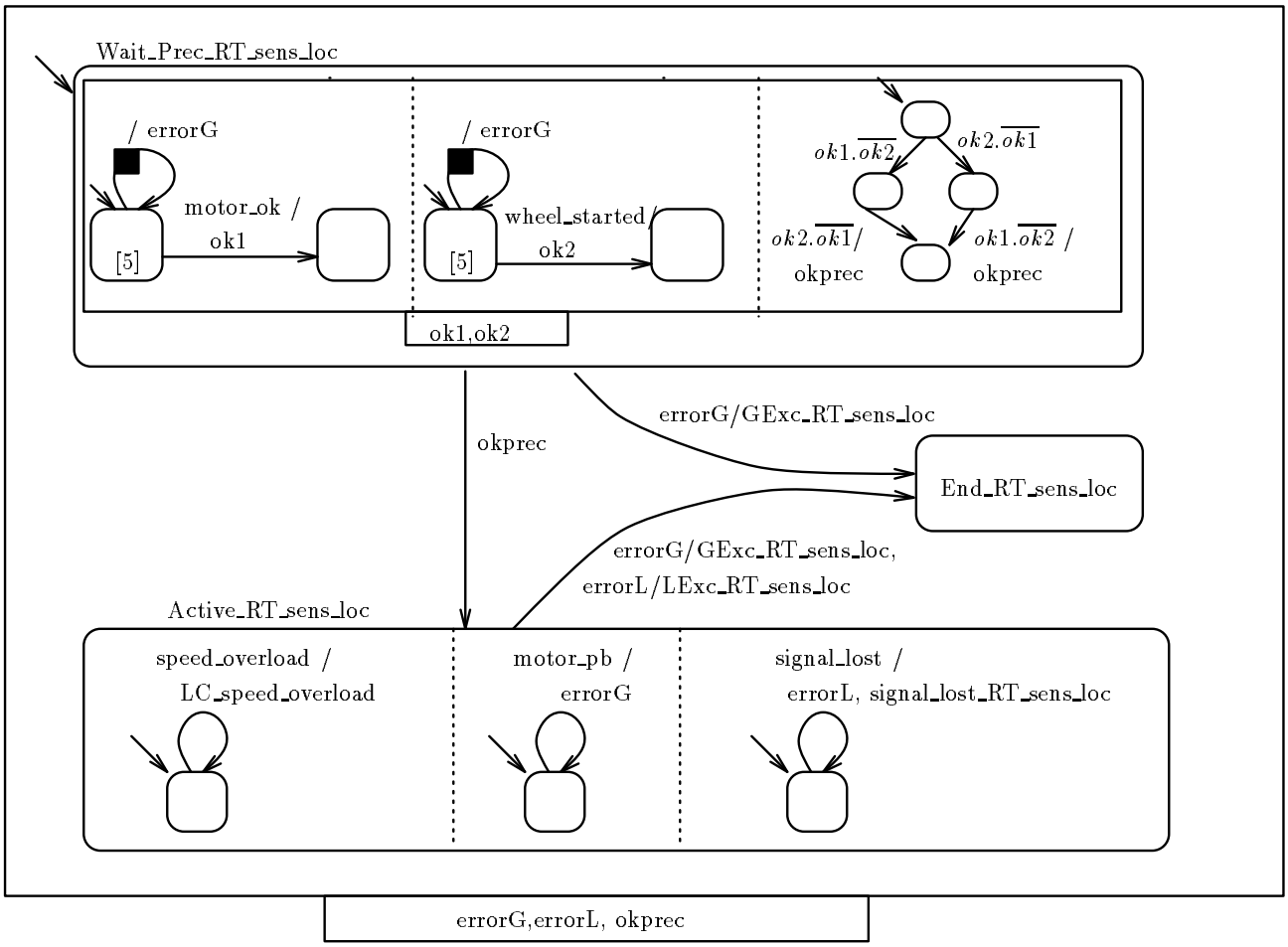

Figure 5: the RT RT_sens_loc specification translated into Timed Argos

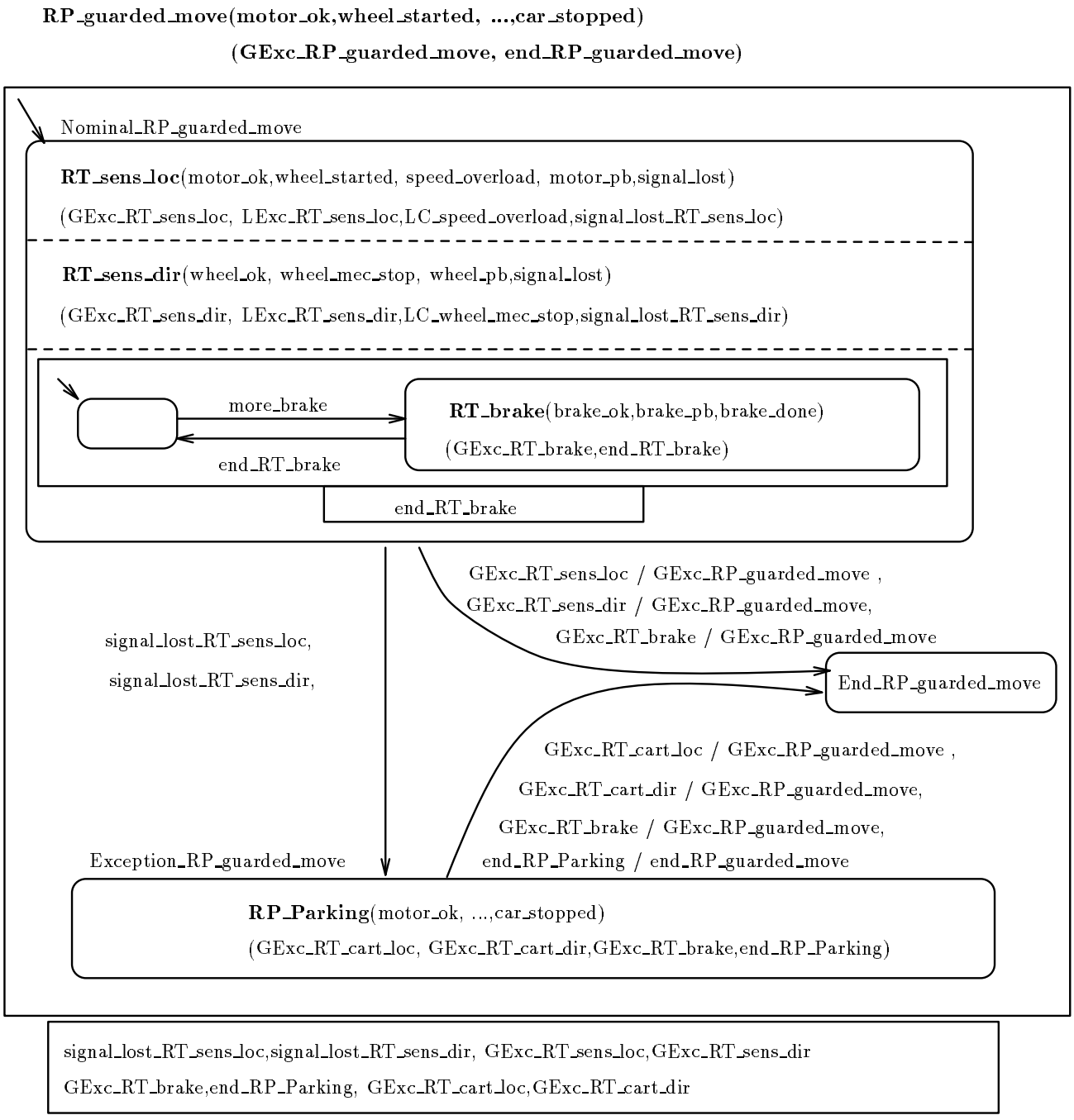

Figure 6: the RP RP_guarded_move specification translated into Timed Argos

(for instance if the nominal execution is a parallel composition of two RTs such that the sum of their execution time is lower than `min_post`).

The TCTL formula which expresses that a RP is time bounded is the following one :

`In_State(Initial(RP))` $\Rightarrow$ $\forall\exists_{\leq T}$ `In_State(End_RP)`

`Initial(RP)` gives the initial state of the Timed Argos program: either Wait_Prec_RP or Nominal_RP, according to the presence or absence of preconditions in the RP specification. This formula must be interprated as follows: each execution sequence issued from the initial state of the Timed Argos program is such that before T units of time the "end state" has been reached.

The second example we take to illustrate our method, consists in proving that the time-lag between the starting point of the wheel execution law and the motor one is bounded. The specification is such that the wheel is already controlled when the execution law of the motor is started, but the reverse is not true. We could check that the time lag between the two starting points is always lower than 6 milli-seconds, due to the waiting time limit associated with the two RT_sens_loc preconditions. In order to express this property, we have to use the following state predicates : `In_State(Active_RT_sens_loc)` and `In_State(Active_RT_sens_dir)`. The appropriate TCTL formula should express that when the state `Active_RT_sens_dir` is active, it is not possible to stay in it for T milli-seconds and then to reach the state `Active_RT_sens_loc`, with $T \geq 6$. This is done by using a TCTL operator :$\phi\ \exists\mathcal{U}\ \phi'$ which is not given in our definition of TCTL formulas because of its complexity for a first approach of this logic. This formula is satisfied if and only if one execution exists such that ϕ is satisfied on each state reached by the sequence before reaching a state q which satisfied ϕ'. Finally, the TCTL formula we are looking for is the following one :

not (`In_State(Active_RT_sens_dir)` $\Rightarrow$
`In_State(Active_RT_sens_dir)` $\exists\mathcal{U}_{=6}$ `In_State(Active_RT_sens_loc)`)

4 Qualitative and quantitative analysis of the Timed-Argos language and environment and the Kronos tool

4.1 Quantitative analysis of both Argos compiler and Kronos tool

The time performances we give in this section have been obtained on a middle-size workstation.

The first quantitative results of the experiments we performed concern the Argos compiler performances. We give in the following table the information about the

timed-graph computed: the number of transitions, states, internal events, clocks and inputs — each of theses quantities is involved in the compilation time of a Timed-Argos program.

The main program is the Timed-Argos program associated with the main PR we previously described. The program named Reduced is slightly different from the main one. It has been obtained by associating with each set of preconditions one waiting time limit, instead of one waiting time limit with each precondition. We will see below that this slight difference of the specification has significant consequences on the performance of both Argos compiler and Kronos tool.

Example	Trans	States	Internal Events	Clocks	Inputs	Compilation Time
Main	31012	105	40	15	30	0h 20
Reduced	12978	105	40	11	30	0h 10

The first remark we can make is that despite the high number of internal events used in these programs, which are known to be responsible for bad time performance in synchronous languages compiler (since communications are statically computed), time performance of the Argos compiler is quite good. This is due to the adaptation of an algorithm used in the Esterel compiler whose complexity is linear in terms of the number of internal events and not exponential as the first algorithms used in the argos compiler.

The second remark is that the Argos compiler is very sensitive to the number of clocks used in the source program. Roughly, each time we introduce a new clock in the source program, we increase twofold the number of transitions of the computed timed graph (in fact, this is exactly the case if the new clock is active in each global state of the source program).

In the next table, we focus on the time performance of the kronos tool while checking that the application is time-bounded.

Example	Clocks	Trans	States	Verification Time
Main	15	31012	105	$>$ 3 h 00
Main minimized	15	4004	105	1 h 50
Main optimized	9	31012	105	$>$ 3 h 00
Main optimized & minimized	9	4004	105	0h 50
Reduced	11	12978	105	0 h 10
Reduced minimized	11	705	44	0 h 01

The first time we try to prove the TCTL formula on the Timed-Graph computed by the Argos compiler from the main program was a failure: after three hours we stop the tool without having any result. The Timed-Graph size was too high for the Kronos tool capacities. This first experiment leads us to find a way to reduce the size of the Timed-Graph computed by the Argos compiler before using Kronos. We experiment two methods. The first one is based on the fact that the property we check do not depend on the Timed-Graph labels. As a consequence, it is possible to abstract them and then to minimize the Timed-Graph. This minimization leads to good results since the behaviors we specify are such that a lot of transitions lead to the same "end" state. Indeed, if we study the specification formalism precisely, a lot of controller reactions consist in reaching the end state. This is the case for the RT when a local or global exception is detected, when a delay expires, ...

In order to achieve this reduction phase we use a tool named Aldebaran, whose time performances are very good. The minimization of the Timed-Graph associated with the main program takes less than 3 minutes. The number of transition decrease by about 85 %. The Kronos tool takes less than 2 hours to give the result of the verification process.

The second method we experiment is not as general as the fisrt one. In some sens it optimized the number of clocks used in the Timed-Graph computed by the Argos Compiler. This one generates a new clock for each delay encountered in the source program. This is not a good idea since some clocks could be re-used for two different delays because the structure of the source program is such that these two delays could not be active in the same global state. For instance, this is obviously the case when the two delays are associated with two states of the same automaton. We apply this optimization phase on the Timed-Graph associated with the main program, we reduce the number of clocks from 15 to 9. It was not sufficient to make possible the use of the Kronos tool on the resulting Timed Graph but after minimization the time performances benefit greatly from this clock optimization.

We also make the same tests for the reduced program (with global waiting time limits). We could see that the Kronos performances are quite good without doing anything on the Timed-Graph obtained by the Argos compiler. The minimization phase of the Timed Graph allows to have better results.

From the quantitative point of view, we can make the following remarks:

- Although the example we implemented is not very complex for the domain area, the size of the Timed-Graph which represents the controller is important: 150 states, 30000 transitions and 15 clocks.

- The Timed-Argos compiler performances are quite good for this Timed-Graph size.
- However, this size of Timed Graph is beyond the capacities of Kronos. We had therefore to find methods allowing to reduce this size before checking a property.

4.2 Qualitative analysis

The first outcomes of the experimentation leads us to express some qualitative issues :

- The Timed Argos language is well-adapted to this application area, since the translation of the Robot-Procedure formalism into a Timed Argos program is quite simple.
- The expressivity of the TCTL logic is sufficient for the properties we would like to prove .
- Nevertheless, it is not possible for an end user in robotics to directly express his properties in TCTL. A more user-friendly formalism, which should take the application specificities into account, has to be defined.
- When a property is not satisfied the diagnosis given by Kronos does not allow to find easily (it is impossible for a non-expert user of Kronos) the error in the program. Indeed, the diagnosis is expressed in terms of Timed-Graph informations and not in terms of the source program. Given the fact that the semantics of Argos is such that it is easy to rely the information model on the source program, this drawback should be disappeared in the future.

5 Conclusion

We have shown in this paper how to use formal methods to check time dependent properties inside an existing development environment for robot controllers. This method is based on Timed Argos, a temporal extension of a synchronous language, and on Kronos, a symbolic verification tool of the real time temporal logic TCTL. Its principle consists in translating the specification formalisms used in the environment in Timed Argos, translating the properties into TCTL formulas, and finally using the Kronos tool to know whether the property is satisfied or not. We illustrate our approach by a real example taken from the area of autonomous vehicles. This

application has been implemented on a real time execution system, and experiments on a "virtual train" of two vehicles have been made with success.

The first conclusion we can draw from our experiments is that the approach we follow is well adapted in its principles to the needs of this application area. However the performances of the used tools must be improved in order, on one hand, to handle *large-scale* real application, and on the other hand, to be integrated within existing development environments. Studying "on the fly" symbolic methods of verification could be a way to fulfil these time requirements. Another possible solution to improve the time performance of the verification process is to define a specific translation of the RP formalism into Timed Graph, which takes in account some characteristics of this formalism. For example, most of the delay expirations lead to the "end state". If we take this information in account while translating a RP into a Timed-Graph, it is not necessary to detail the status of the other delays and the number of transitions would be dramatically reduced.

We are currently testing a new algorithm which has been implemented in Kronos to compute whether a global state is accessible or not. This algorithm is supposed to have good time performance. It is possible to transform the Timed Argos program associated with a RP by adding in parallel a "synchronous observer" as defined in [HLR93], in such a way that the satisfaction of a property is equivalent to the unreachability of a particular error state. The problem encountered when using this method is the following: if a delay is present in the observer component (this is the case if the property is time dependent) the number of transition number of the Timed Graph associated with the program is increased twofold since this delay is always active. Thus the compilation time of this Argos program is increased and the static analysis of the Timed Graph performed by Kronos before evaluating the property also takes more time and finally the results are less interesting than expected.

Moreover, an important work remains to be done in order to make easier both property specification and diagnosis interpretation. It will be more interesting in the future to express properties by using a formalism more intuitive than TCTL. This one should take into account the application characteristics and should be automatically translated into TCTL formulas. More precisely, our idea is to identify a set of interesting generic properties that the programmer could easily use for specific cases. For instance, a property which seems to be required most of the time is that the robotic application is time-bounded. In the solution we propose, the user has only to specify the time bound. An automatic translation into TCTL allows to check whether the property is satisfied or not.

We have already identified two other generic properties, with which we can check that :

- the time-lag between the starting point of two sequential execution laws is bounded;
- the time-lag between the starting point of two overlapping execution laws is bounded;

These two types of properties must be distinguished since their translation into TCTL formulas are different.

Finally, these experiments also show us a new interesting research direction for this kind of applications. The method we experiments here provides the user with a way to verify if its program satisfies a time dependent constraint, but from the designer's point of view nothing is done to help him to select the right delay values which will necessarily satisfy this constraint. This complementary problem is indeed very relevant for the application area. We are thus starting the evaluation of a candidate method to address this problem It is based on another extension of Argos to hybrid systems and on a tool for synthesizing linear invariants named polka [HPR94]. The first tests we performed on a RT were successful. However, problems of time performance should arise if the number of unvalued clock is too high.

Acknowledgements

I would like to thank Sergio Yovine and Conrado Daws for their help during the use of the Kronos tool.

Appendix

RT_brake is defined by :
`Name` : RT_brake
`Preconditions` : brake_ok [`waiting time limit :` 5 ms]
`Postconditions` : brake_done [`waiting time limit :` 5 s]
`Duration` : `undefined`
`Global exceptions` : brake_pb
`Local exceptions` : none
`Local changes` : brake_mec_stop

In this specification:

- brake_ok indicates that the foot_brake initialization has been made without detecting errors;
- brake_done occurs when the engine braking is able to satisfy the braking request without using the foot_brake.
- brake_pb occurs when a mechanical problem is detected in the foot_brake;
- brake_mec_stop indicates that the foot_brake reaches a mechanical stop.

The RP RP_parking is defined by :

```
Name : RP_parking
Preconditions : none
Postconditions : car_stopped [waiting time limit : 1 m]
Global exceptions : none
Local exceptions : none
Nominal execution :
              Parallel
                   start(RT_cart_loc)
                   ||
                   start(RT_cart_dir)
                   ||
                   start(RT_brake)
              endParallel
Exception treatments : none
```

The RT named RT_cart_loc (resp. RT_cart_dir) controls the electric motor (resp. the wheel) of the car according to a well-suited trajectory, which has been already computed. They are not detailed here.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the fifth annual IEEE symposium on Logics In Computer Science*, pages 414–425, Philadelphie, PA, USA, June 1990.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *Proceedings of ICALP 90*, pages 322–335. Springer Verlag, 1990.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[dSV89] R. de Simone and D. Vergamini. Aboard auto. Rapport Technique 111, INRIA, 1989.

[Fer88] J.C. Fernandez. Aldébaran, a tool for verification of communicating processes. Technical report spectre 14, LGI-IMAG, 1988.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente*. Workshops in Computing, Springer Verlag, June 1993.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*. IEEE Computer Society Press, June 1992.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (belgium), september 1994. LNCS 864, Springer Verlag.

[JM94] M. Jourdan and F. Maraninchi. A modular state/transition approach for programming reactive systems. In *Worshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, June 1994.

[JMO93] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *5th International Conference on Computer-aided Verfication*, Elounda, June 1993. LNCS 697, Springer Verlag.

[Kap94] K. Kapellos. Environnement de programmation des applications robotiques re'actives, 1994.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992.

[PDDM94] M. Parent, P. Daviet, J.C. Denis, and T. M'Saada. Automatic Driving inStop and Go Traffic. In *Proceedings of the Conference IEEE Intelligent Vehicles*, 1994.

[SECK93] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computer-aided design of a generic robot controller handling reactivity and real-time control issues. In *IEEE Transactions on Control Systems Technology, vol. 1, no 4*, December 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399